Clase 4 — 20.11.25

👨‍🏫 Profesor: José Antonio Martín
📘 Unidad: Programación Multiproceso
🗓️ Clase 4 — 20/11/2025
🎯 Tema: Finalización de la Programación Multiproceso y Gestión de Servicios

---

2️⃣7️⃣ PROCESOS EN LINUX

En Linux, un **proceso** es un programa en ejecución junto con **todo su contexto de ejecución**, que es la información que el kernel necesita para poder gestionarlo correctamente en un entorno multiproceso. Este contexto no se limita al código del programa, sino que incluye elementos fundamentales como:

- El **PID** (identificador del proceso).
- El estado del proceso (ejecutándose, dormido, detenido, etc.).
- La memoria que tiene asignada (heap, stack, segmentos compartidos).
- Los **descriptores de fichero** abiertos (archivos, sockets, pipes).
- Las variables de entorno.
- La relación con otros procesos (padre e hijos).
- Las señales que puede recibir o gestionar.

Linux está diseñado para que el multiproceso sea **transparente y observable**. A diferencia de otros sistemas operativos, no oculta estos detalles, lo que permite entender de forma clara cómo el sistema reparte el tiempo de CPU entre múltiples tareas.

Cada proceso existe porque el **kernel** lo ha creado y lo mantiene dentro de sus estructuras internas. El kernel decide:

- cuándo un proceso se ejecuta,
- durante cuánto tiempo,
- cuándo se suspende,
- y cuándo se reanuda.

Gracias a esta planificación, el sistema puede ejecutar **muchos más procesos que núcleos físicos**, dando la sensación de ejecución simultánea.

PID: Identificador del proceso

Todo proceso activo en Linux tiene asignado un **PID (Process ID)**, que es un número entero único mientras el proceso existe. El PID es la forma básica que tiene el sistema de **referirse a un proceso concreto**.

El PID permite:

- Identificar el proceso de forma inequívoca.
- Monitorizar su consumo de CPU y memoria.
- Enviarle señales (terminar, pausar, reiniciar).
- Establecer relaciones entre procesos.

Es importante entender dos ideas clave:

1. **El PID no es permanente**
   Cuando un proceso termina, su PID puede ser reutilizado por el sistema para otro proceso distinto.
2. **Los procesos están organizados jerárquicamente**
   Cada proceso suele tener un **proceso padre (PPID)**, formando una estructura en forma de árbol.

Esta jerarquía se puede observar con:

```
ps axjf
```

Este comando muestra:

- El árbol de procesos.
- Qué proceso ha lanzado a cuál.
- La relación padre-hijo entre procesos.

Gracias a esta estructura se entiende, por ejemplo, por qué al cerrar una aplicación gráfica se finalizan **varios procesos a la vez**: se está cerrando una rama completa del árbol.

Visualización de procesos: `ps`

El comando `ps` (process status) permite visualizar los procesos del sistema, pero es fundamental entender que **no es un monitor en tiempo real**. Cada vez que se ejecuta, muestra una **instantánea** del sistema en ese momento exacto.

Un uso muy habitual es:

```
ps aux
```

Donde:

- `a` → muestra procesos de todos los usuarios.
- `u` → formato orientado a usuario (CPU, memoria, etc.).
- `x` → incluye procesos sin terminal asociada (servicios y daemons).

Este comando muestra información como:

- Usuario propietario del proceso.
- PID.
- Porcentaje de CPU y memoria.
- Estado del proceso.
- Tiempo de ejecución.
- Comando que originó el proceso.

Desde un punto de vista conceptual, `ps` responde a la pregunta:

> **“¿Qué procesos existen ahora mismo en el sistema?”**

Otros usos importantes de `ps` incluyen el filtrado, por ejemplo:

```
ps aux | grep bash
```

Esto permite localizar procesos concretos combinando `ps` con herramientas como `grep`.

Estados de un proceso

Un proceso en Linux puede encontrarse en distintos estados, visibles en `ps` o `top`:

- **Running (R)** → ejecutándose en CPU.
- **Sleeping (S)** → esperando algún recurso.
- **Stopped (T)** → detenido manualmente.
- **Zombie (Z)** → proceso finalizado cuyo padre no ha recogido su estado.

Los procesos *zombie* no consumen CPU, pero indican un problema de gestión por parte del proceso padre.

Monitorización en tiempo real: `top`

A diferencia de `ps`, el comando `top` ofrece una **visión dinámica y en tiempo real** del sistema. Es una herramienta clave para entender cómo se comportan los procesos mientras el sistema está en funcionamiento.

Al ejecutar:

```
top
```

podemos observar:

- Cambios continuos en el uso de CPU.
- Cambios en el consumo de memoria.
- Procesos que aparecen y desaparecen.
- La **carga media del sistema (load average)**.

La carga media indica cuántos procesos están esperando CPU en distintos intervalos de tiempo, lo que permite evaluar si el sistema está saturado.

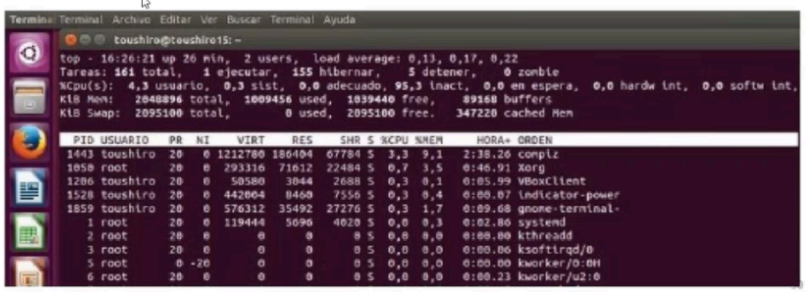
Intervalo de refresco en `top` (IMPORTANTE)

El comando:

```
top -d 5
```

indica que la información se actualiza cada **5 segundos**.

top –d 5 (Donde 5 es el número de segundos a transcurrir entre cada muestreo)
top –o %CPU (Donde %CPU es el valor por el que vamos a ordenar los procesos)



Esto es importante porque:

- Intervalos muy cortos generan más consumo y ruido visual.
- Intervalos largos pueden ocultar picos breves de actividad.

Este parámetro permite adaptar `top` a análisis rápidos o a observaciones prolongadas.

Ordenación de procesos en `top`

`top` permite ordenar los procesos por distintos criterios. Por ejemplo, por consumo de CPU:

```
top -o %CPU
```

O por memoria:

```
top -o %MEM
```

Esto facilita detectar:

- Procesos que consumen demasiada CPU.
- Procesos con fugas de memoria.
- Aplicaciones mal comportadas.

`htop`: gestor de procesos avanzado

`htop` es una versión mejorada de `top` que añade:

- Representación visual de CPU y memoria.
- Colores.
- Navegación con teclado.
- Acciones directas sobre procesos.

Una ventaja importante es que permite:

- Seleccionar un proceso.
- Enviarle una señal.
- Cambiar su prioridad.

Todo ello sin necesidad de conocer previamente su PID.

Control de procesos: `kill`, señales y permisos

El comando `kill` se utiliza para **enviar señales** a los procesos. El nombre puede llevar a error: no siempre implica matar el proceso.

Ejemplo:

```
kill -9 3484
```

Aquí:

- `kill` envía una señal al kernel.
- `-9` corresponde a **SIGKILL**.
- `3484` es el PID del proceso.

SIGKILL fuerza la finalización inmediata del proceso. El kernel lo elimina sin dar opción al programa de cerrar correctamente.

Otras señales importantes incluyen:

- **SIGTERM (15)** → cierre ordenado (por defecto).
- **SIGSTOP** → detiene el proceso.
- **SIGCONT** → reanuda un proceso detenido.

Permisos al usar `kill`

Linux aplica un control estricto de permisos:

- Un usuario solo puede enviar señales a **sus propios procesos**.
- Para controlar procesos de otros usuarios o del sistema es necesario **root**.

Por eso muchos comandos de administración requieren `sudo`.

`pkill` y `killall`

Para evitar trabajar con PIDs, Linux ofrece comandos basados en nombres de procesos.

`pkill`

Permite matar procesos por nombre:

```
pkill -9 htop
```

Este comando enviará la señal a todos los procesos cuyo nombre coincida.

`killall`

Envía una señal a **todos los procesos de un programa concreto**:

```
killall firefox
```

Es un comando potente y rápido, pero peligroso si no se usa con cuidado, ya que puede cerrar múltiples procesos simultáneamente.

2️⃣8️⃣ PROCESOS EN PRIMER Y SEGUNDO PLANO

Cuando trabajamos en Linux desde la terminal, los procesos no solo se diferencian por lo que hacen, sino también por **cómo se relacionan con la sesión de usuario**. Aquí entra en juego la distinción entre **procesos en primer plano (*foreground*)** y **procesos en segundo plano (*background*)**, un concepto fundamental para entender el uso real de la terminal.

Esta diferencia no es estética ni secundaria: determina **quién controla el teclado**, **quién escribe en pantalla** y **si el usuario puede seguir trabajando** mientras un proceso está activo.

Proceso en primer plano (*foreground*)

Cuando ejecutamos un comando de forma normal en la terminal:

```
nombreAplicacion
```

el proceso se lanza en **primer plano** por defecto.

Esto implica varias cosas importantes:

- El proceso **toma el control completo de la terminal**.
- El usuario **no puede ejecutar ningún otro comando** mientras el proceso siga activo.
- La entrada estándar (teclado) y la salida estándar (pantalla) quedan **asociadas exclusivamente a ese proceso**.
- La terminal queda bloqueada hasta que el proceso termina o se interrumpe.

Este comportamiento es totalmente intencionado. El *foreground* permite que el usuario **interactúe directamente con el proceso**, enviándole datos por teclado o recibiendo información por pantalla.

Ejemplos típicos de procesos en foreground:

- Programas interactivos.
- Scripts que piden datos al usuario.
- Comandos que muestran resultados directamente en pantalla.

👉 Es importante entender que **un proceso en foreground no es un proceso "mal ejecutado"**. Es el comportamiento normal y esperado.

Interrupción de un proceso en primer plano

Si un proceso en primer plano se queda bloqueado, tarda demasiado o simplemente queremos recuperar el control de la terminal, podemos usar:

```
Ctrl + Z
```

Esta combinación de teclas envía una señal especial al proceso.

El efecto es el siguiente:

- El proceso **no se cierra**.
- El proceso pasa a segundo plano en estado **stopped** (detenido).
- La terminal queda libre para el usuario.

Este punto es clave:
detener un proceso no es lo mismo que matarlo. El proceso sigue existiendo en memoria, pero no se está ejecutando.

Este mecanismo es muy útil cuando:

- Un comando tarda más de lo esperado.
- Un proceso interactivo deja de responder.
- Queremos posponer la ejecución sin perder el estado del proceso.

Proceso en segundo plano (*background*)

Un proceso en segundo plano se ejecuta **sin bloquear la terminal**, permitiendo al usuario seguir trabajando mientras el proceso continúa activo.

Para lanzar un proceso directamente en segundo plano se utiliza el carácter `&` al final del comando:

```
nombreAplicacion &
```

Cuando se hace esto:

- El proceso se inicia y continúa ejecutándose.
- La terminal devuelve inmediatamente el control al usuario.
- El proceso sigue consumiendo CPU, memoria y otros recursos.
- El proceso no recibe entrada directa del teclado (salvo redirecciones).

Este tipo de ejecución es habitual en:

- Scripts largos.
- Procesos auxiliares.
- Programas que no requieren interacción constante.

Desde el punto de vista del sistema, **un proceso en background no es menos importante que uno en foreground**: la diferencia está únicamente en la relación con la terminal.

Relación entre foreground y background

Un mismo proceso puede cambiar de estado a lo largo de su vida:

- Puede iniciarse en foreground.
- Ser detenido con `Ctrl + Z`.
- Quedar en background en estado *stopped*.
- Volver a ejecutarse posteriormente.

Esto demuestra que *foreground* y *background* **no son tipos de procesos distintos**, sino **modos de ejecución** asociados a la sesión del usuario.

Comando `jobs`: control de procesos de la sesión

Para ver los procesos que están asociados a la terminal actual se utiliza:

```
jobs
```

Este comando muestra:

- Los procesos lanzados desde esa sesión.
- Si están ejecutándose en background.
- Si están detenidos (*stopped*).

Es importante destacar que:

- `jobs` **no muestra todos los procesos del sistema**.
- Solo muestra los procesos controlados por la **shell actual**.

Por eso, `jobs` complementa a `ps` y `top`, pero no los sustituye.

Foreground y background desde el punto de vista del sistema

Desde el kernel, un proceso en foreground y uno en background son prácticamente iguales. La diferencia principal es:

- A qué proceso está asociada la **terminal**.
- Qué proceso recibe las señales del teclado.

Esto explica por qué:

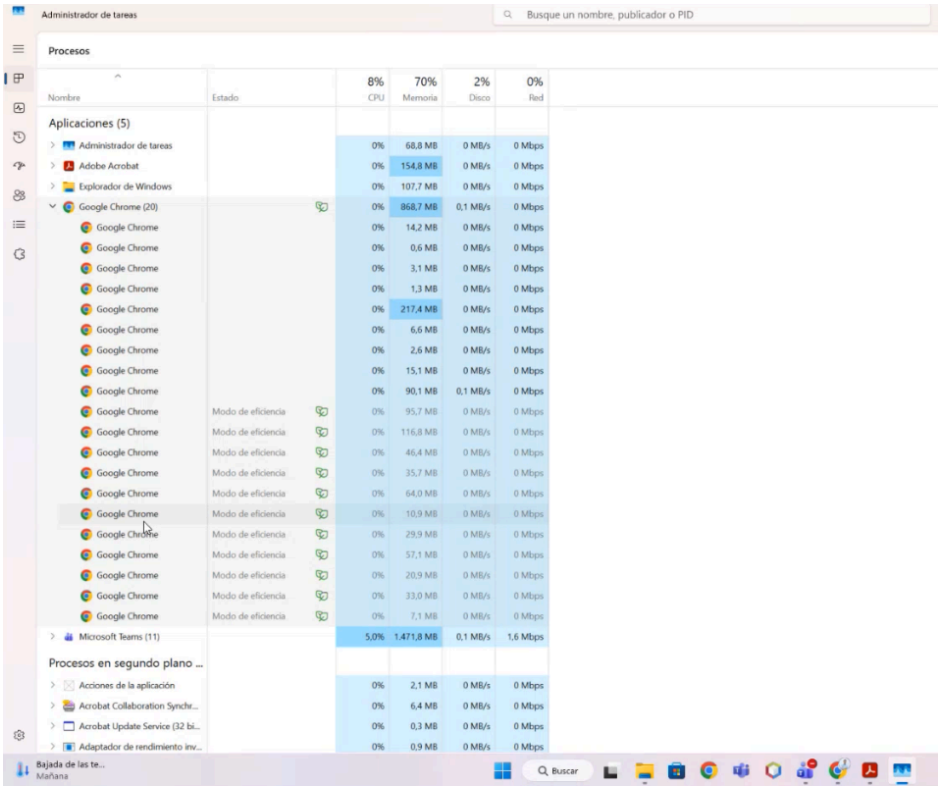
- Un proceso en background puede seguir funcionando incluso si el usuario no interactúa.
- Un proceso en foreground puede bloquear completamente la sesión.

Comparación con Windows

En Windows, estos conceptos existen, pero están **ocultos al usuario final**. El sistema gestiona automáticamente qué procesos están en primer plano y cuáles en segundo plano.

Ejemplos habituales:

- Google Chrome ejecuta múltiples procesos en segundo plano.



- Microsoft Teams mantiene servicios activos aunque la ventana no esté en uso.
- Servicios del sistema se ejecutan siempre en background.

Linux no hace nada diferente a nivel conceptual. La diferencia es que **Linux expone estos mecanismos directamente al usuario**, permitiendo un control mucho más fino desde la terminal.

2️⃣9️⃣ SERVICIOS

Un **servicio** en Linux es un tipo especial de proceso cuya finalidad **no es interactuar directamente con el usuario**, sino **proporcionar funcionalidad al sistema o a otros programas**. Por este motivo, los servicios **no muestran ventanas**, **no reciben entrada por teclado** y **no dependen de una sesión de usuario concreta**.

Normalmente, los servicios se ejecutan en **segundo plano** (*background*) y permanecen activos durante largos periodos de tiempo, a menudo desde que el sistema arranca hasta que se apaga.

Ejemplos típicos de servicios son:

- Servicios de red.
- Firewalls.
- Servidores web.
- Servicios de actualización.
- Servicios de autenticación o registro de eventos.

Desde el punto de vista del sistema operativo, un servicio **sigue siendo un proceso**, con su PID, su consumo de memoria y CPU, y su ciclo de vida propio. La diferencia no está en *qué es*, sino en *para qué está diseñado*.

Relación entre procesos normales y servicios

La principal diferencia entre un proceso lanzado por el usuario y un servicio es **cómo se inicia y cómo se gestiona**:

- Un proceso normal suele iniciarse manualmente por un usuario.
- Un servicio suele iniciarse automáticamente por el sistema.
- Un proceso normal suele terminar al cerrar la sesión.
- Un servicio continúa ejecutándose aunque no haya ningún usuario conectado.

Por eso, los servicios son fundamentales para el funcionamiento interno del sistema.

Tipos de inicio de un servicio

Un servicio puede configurarse para iniciarse de distintas formas. Este apartado es **especialmente importante** a nivel práctico y de examen.

Inicio automático

Un servicio configurado como **automático**:

- Se inicia al arrancar el sistema operativo.
- No necesita intervención del usuario.
- Se utiliza para servicios críticos o necesarios para el funcionamiento básico del sistema.

Aunque sea automático, **puede detenerse manualmente** si el administrador lo decide.

Inicio manual (MUY IMPORTANTE)

Un servicio configurado como **manual**:

- **No se inicia automáticamente** con el sistema.
- Puede iniciarse cuando el administrador lo necesite.
- **Permite iniciar, detener y reiniciar el servicio manualmente**.

Este modo es clave en:

- Entornos de pruebas.
- Servicios que no siempre son necesarios.

- Tareas de mantenimiento y diagnóstico.

👉 El profesor insiste en este punto:
el modo manual permite detener y reiniciar el servicio, lo cual lo diferencia claramente del modo deshabilitado.

Servicio deshabilitado

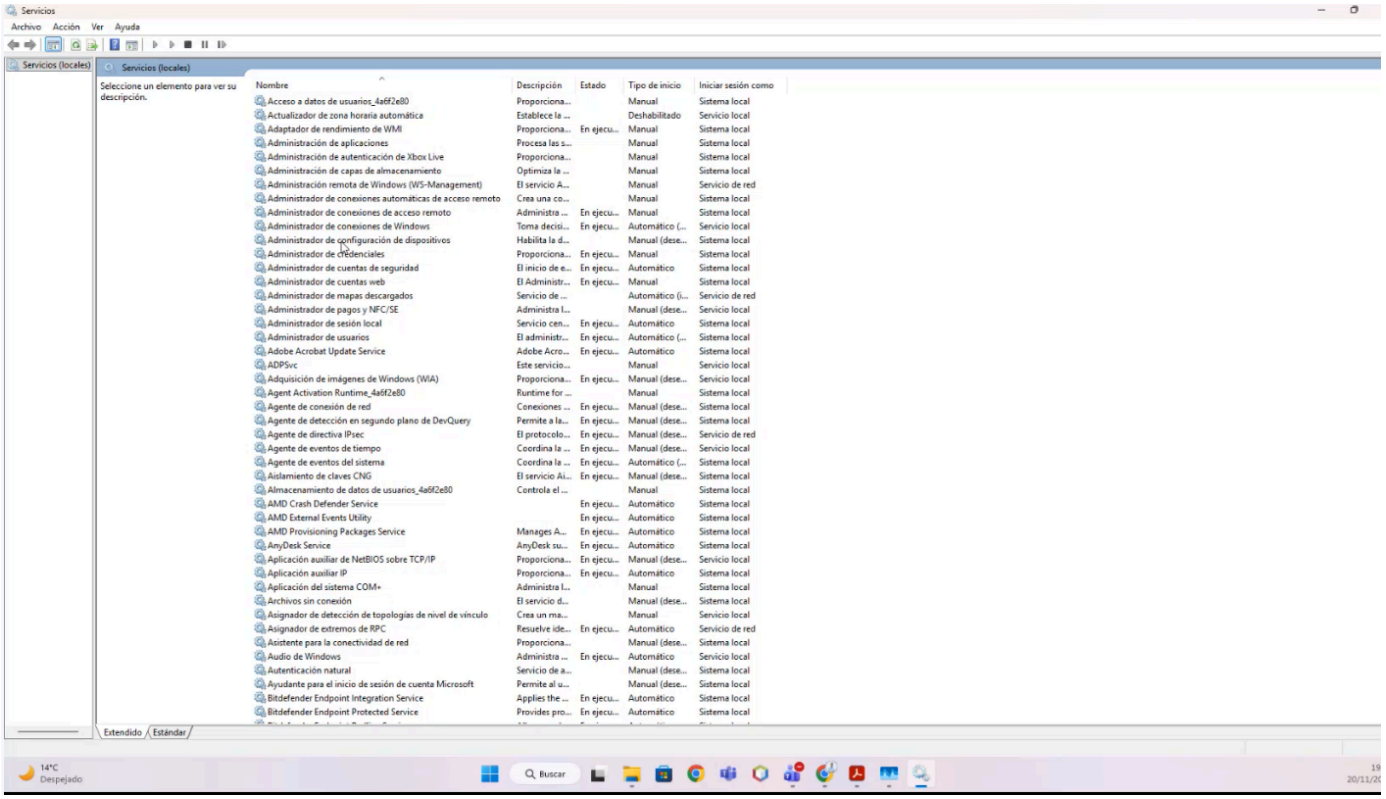
Un servicio deshabilitado:

- **No puede iniciarse**, ni automáticamente ni manualmente.
- El sistema impide su ejecución.
- Se utiliza para servicios innecesarios o que podrían suponer un riesgo.

Un servicio deshabilitado existe en el sistema, pero está completamente inactivo.

Estados de un servicio

Independientemente del tipo de inicio, un servicio puede encontrarse en distintos **estados de ejecución**.



Los estados básicos son:

- **Ejecutándose**
  El servicio está activo y funcionando.
- **Detenido**
  El servicio no está en ejecución, pero puede iniciarse (si no está deshabilitado).

Es importante entender que:

- Un servicio automático **puede estar detenido**.
- Un servicio manual **puede estar ejecutándose**.

El tipo de inicio indica *cómo se inicia*, no *si está activo en este momento*.

Servicios en Linux: systemd

En Ubuntu y en la mayoría de distribuciones Linux modernas, la gestión de servicios se realiza mediante **systemd**, que es el sistema de inicialización y gestión de servicios actual.

`systemd` se encarga de:

- Iniciar servicios al arrancar el sistema.
- Detenerlos correctamente.
- Reiniciarlos cuando sea necesario.
- Registrar su estado y su historial de ejecución.

Comandos básicos con systemd

Todos los comandos de gestión de servicios requieren **privilegios de administrador**, ya que afectan al funcionamiento del sistema.

Detener un servicio

```
sudo systemctl stop ufw
```

Finaliza la ejecución del servicio indicado.

Iniciar un servicio

```
sudo systemctl start ufw
```

Arranca el servicio manualmente.

Reiniciar un servicio

```
sudo systemctl restart ufw
```

Detiene y vuelve a iniciar el servicio en un solo paso. Muy habitual tras cambios de configuración.

Consultar el estado de un servicio

```
sudo service ufw status
```

Este comando muestra:

- Si el servicio está activo o detenido.
- Desde cuándo se está ejecutando.
- Mensajes recientes relacionados con su estado.

Consultar el estado es una práctica básica antes y después de realizar cambios.

Eliminar un servicio

Eliminar un servicio no consiste únicamente en detenerlo. Para hacerlo correctamente hay que seguir una secuencia lógica:

1️⃣ Detener el servicio

```
systemctl stop service-name
```

Esto asegura que el servicio no esté en ejecución.

2️⃣ Desactivar el servicio

```
systemctl disable service-name
```

Esto impide que el servicio se inicie automáticamente en el futuro.

3️⃣ Eliminar los archivos de configuración

Los servicios gestionados por systemd tienen archivos de configuración que deben eliminarse para evitar referencias residuales:

```
rm /etc/systemd/system/service-name
rm /etc/systemd/system/service-name/[related symlinks]
```

De esta forma se garantiza que el servicio no vuelva a aparecer ni a ejecutarse.

SysV init y Upstart (contexto histórico)

Antes de la adopción de systemd, Ubuntu utilizó otros sistemas de inicialización:

- **SysV init**
- **Upstart** (entre Ubuntu 9.10 y 14.10)

Estos sistemas utilizaban scripts y archivos de configuración ubicados en:

```
/etc/init
```

Aunque hoy están obsoletos, siguen apareciendo en:

- Sistemas antiguos.
- Documentación heredada.
- Entornos legacy.

Conocerlos permite entender comandos y configuraciones antiguas que aún pueden encontrarse.

Resumen de la unidad

La programación multiproceso y la gestión de servicios permiten al sistema:

- Aprovechar mejor los recursos de la CPU.
- Ejecutar múltiples tareas de forma simultánea.
- Mantener servicios activos sin intervención del usuario.

- Aumentar la estabilidad y la robustez del sistema.
- Aislar procesos para mejorar la seguridad.

Siguiente unidad

La siguiente clase introduce **programación multihilo**, donde varios hilos se ejecutan dentro de un mismo proceso, compartiendo memoria y recursos. Este enfoque mejora el rendimiento, pero introduce nuevos retos relacionados con la sincronización y la concurrencia.
